GMCWare: A Greedy Modularity Community-based Simplification Algorithm for Malware Detection

Xingyuan Wei[1, 2], Zijun Cheng[4], Ning Li[1], Congying Liu[2], Yan Wang[1, 2], Degang Sun[2, 3]
Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China[1]
University of Chinese Academy of Sciences, Beijing, China[2]
Computer Network Information Center, Chinese Academy of Sciences, Beijing, China[3]
Space Engineering University[4]
talentedyuan@gmail.com[1st], cheng_monk@163.com[2nd], wangyan@iie.ac.cn[5th]
Corresponding Author: Ning Li, Email: lining01@iie.ac.cn

***Abstract*—With the widespread adoption of Android devices, the number of malware instances continues to rise, posing severe threats to users' security and privacy. Existing malware detection methods primarily rely on static or dynamic analysis, typically performing machine learning classification by extracting features such as permissions or sensitive API calls. However, these methods fail to adequately capture the behavioral patterns of malware and face efficiency bottlenecks when dealing with complex data structures like Function Call Graphs (FCG). To address this, we propose a malware detection method based on the Behavior Relation Graph (BRG). By simplifying the Function Call Graph using community detection algorithms, we partition the complex graph structure into several communities, thereby extracting the functional modules and behavioral patterns of malware and significantly reducing the complexity of analysis. Subsequently, we vectorize the BRG using Graph Neural Networks (GNNs) and combine it with a classification model to achieve high-performance detection and interpretation of malware. Experimental results demonstrate that our method performs excellently on a large-scale dataset containing 12,408 benign software samples and 7,658 malware samples, with an average of 44,704 function nodes. We achieved a classification performance of 100% accuracy and a 100% F1-Score, significantly improving detection accuracy and providing an efficient and reliable solution for malware analysis.**

***Index Terms*—Android Malware Detection, Community Detection Algorithm, GNNs**

I. Introduction

Android malware poses an increasingly significant threat in the digital realm. According to reports from AVTEST, as of January 2025, there were a total of 35,641,466 Android malware samples [1]. The Android operating system accounts for a substantial proportion of mobile devices, resulting in a prolific and large number of Android malware instances. As humans are now more reliant on smartphones than ever before, any security and privacy threats can have catastrophic impacts. Therefore, the detection and study of Android malware are of paramount importance.

Existing malware analysis primarily includes three methods: static analysis, dynamic analysis, and hybrid analysis [2], [3]. Methods based on feature extraction such as permissions, like NATICUSdroid [4] and DREBIN [5], extract sensitive permissions and certain sensitive API call features, then utilize machine learning techniques for data fitting. Although these methods have achieved certain levels of effectiveness, they do not consider the malicious behaviors of programs, which is detrimental to the analysis of malware and its families. In recent years, due to the rapid development of Graph Neural Networks (GNNs), GNNs have been employed for malware analysis [6], [7]. However, the function call graphs used in these methods are often too large and complex, and the representation methods impact efficiency.

We have the intuition that the behaviors of malware authors are reflected in the behaviors of the malware itself. These behaviors often involve altering obvious coding features to evade detection systems, but inadvertent coding habits cannot be erased. These habits are reflected in the malware's behaviors. Function Call Graphs (FCGs) describe the call relationships between functions within software. By applying community detection algorithms to analyze the malware's function call graphs (FCGs), we help identify and understand the behavioral patterns and functional modules of malware. By applying community detection algorithms to function call graphs, the graph can be partitioned into multiple communities, each typically corresponding to specific functional modules or behavioral units. A community represents the smallest unit of behavior, and multiple behaviors constitute the operational logic of the software.

In this study, we first decompiled to obtain the function call graph, then utilized the Greedy Modularity Community Algorithm to simplify the function call graph. The simplified graph obtained through the community algorithm is referred to as the Behavior Relation Graph (BRG), which effectively captures functional modules and behavioral patterns in the malware's graph structure. We then used GNN models to vectorize the BRG, and finally inputted these vectors into a classification model to determine whether the malware is benign or malicious, providing certain explanations to assist in the judgment.

The contributions of this paper are as follows:

1) We developed the malware detection framework GMCWare and improved the traditional Function Call Graph (FCG) method for malware detection by introducing

Thanks to the Chinese Academy of Science, Project E4V01511G3.

the concept of the Behavior Relation Graph (BRG). By simplifying the complex function call graph into several communities, we significantly reduced the complexity of analysis. Especially when dealing with large-scale graph data, community partitioning facilitates distributed computing and parallel processing, thereby enhancing processing efficiency.

2) We extracted instruction features based on predefined categories, identified important instructions by counting the frequency of each category, and then represented these important instructions using BERT [8], thereby improving the effectiveness and efficiency of node representations.
3) We conducted a large-scale evaluation using 12,408 benign software samples and 7,658 malware samples, demonstrating that the GMCWare framework is effective and reliable. This validated that while reducing the size of the Function Call Graph and extracting the Behavior Relation Graph, the effectiveness of malware detection is also enhanced. Finally, our code is open-sourced[1].

II. Related Work

A. Dynamic analysis and Static analysis

While focusing on learning-based malware detection methods, we observe that malware analysis techniques generally fall into three main categories: first, static analysis; second, dynamic analysis; and lastly, hybrid analysis [2]. Static analysis involves using features extracted from the static code analysis process. For example, by analyzing the source code of files (such as Android apk files and Windows PE files) [9], bytecode or binary code [10], requested permissions [11], and opcode sequences [12]. Due to its non-executing nature, static analysis has advantages in speed and safety, avoiding the risk of activating potentially malicious code. However, its effectiveness is somewhat limited because it may not capture all malware variants, especially those that exhibit malicious behavior only during execution.

Another approach is dynamic analysis. The target program runs in a controlled sandbox environment, where various aspects such as the program's API call sequences [3], system calls [13], network traffic [14] [15], and certain data flows during software runtime [16] can be monitored to reveal the program's actual behavior, making this approach closer to real-world conditions. However, dynamic analysis is not widely used because it is too slow, requires a large number of test cases to comprehensively cover program functionalities, and demands excessive resources.

Hybrid analysis combines the aforementioned two techniques, integrating the rapid and comprehensive review capabilities of static analysis with the actual runtime observation capabilities of dynamic analysis. Through static analysis preprocessing, dynamic analysis integration, and result fusion, precise analysis is ultimately achieved. However, hybrid analysis is complex to implement and requires advanced analysis tools and more resources [17], [18], [19], [20]. Due to the characteristics of static analysis—being fast, relatively accurate, and consuming fewer resources—this paper adopts static analysis as its analysis method.

[1]https://github.com/XingYuanWei/GMCWare.git

B. Graph in malware analysis

Malware analysis is increasingly adopting graph-based methods because these approaches can deeply reveal function call flows and recurring patterns within executable files. Researchers now utilize various types of graphs in combination with machine learning (ML) and deep learning techniques for analysis. For example, Control Flow Graphs (CFGs) [21]–[23] illustrate the flow relationships between code basic blocks, identifying malicious behaviors by capturing execution paths and managing data transfers within malicious files. Additionally, API call graphs [24], [25] and function call graphs [26] are used to perform static analysis. Graph representation learning has emerged as a promising method capable of capturing complex patterns in malware programs represented as graphs. In fact, an increasing number of fields are benefiting from these graph-based learning methods and achieving state-of-the-art results.

C. Community Detection Algorithm

Community Detection Algorithms are important tools in graph theory, aimed at identifying the clustering structures or modules of nodes within a graph. These algorithms partition the graph into several communities by analyzing the connectivity density between nodes, where the nodes within a community are densely connected, and the connections between communities are relatively sparse. Common community detection algorithms include the Greedy Modularity Community Algorithm [27], Louvain Algorithm [28], and others. These algorithms have been widely applied in various fields, including information retrieval and malware detection. These algorithms have been widely applied in various fields such as information retrieval and malware detection. The Greedy Modularity Community Algorithm can identify relatively stable and high-quality community divisions, making it suitable for analyzing small to medium-sized networks. As shown in Table II, the graph in this paper has an average node count of 44,704. Since the network scale is not excessively large, we chose commonly used community detection algorithms, including the Greedy Modularity Community Algorithm, for optimization in this paper.

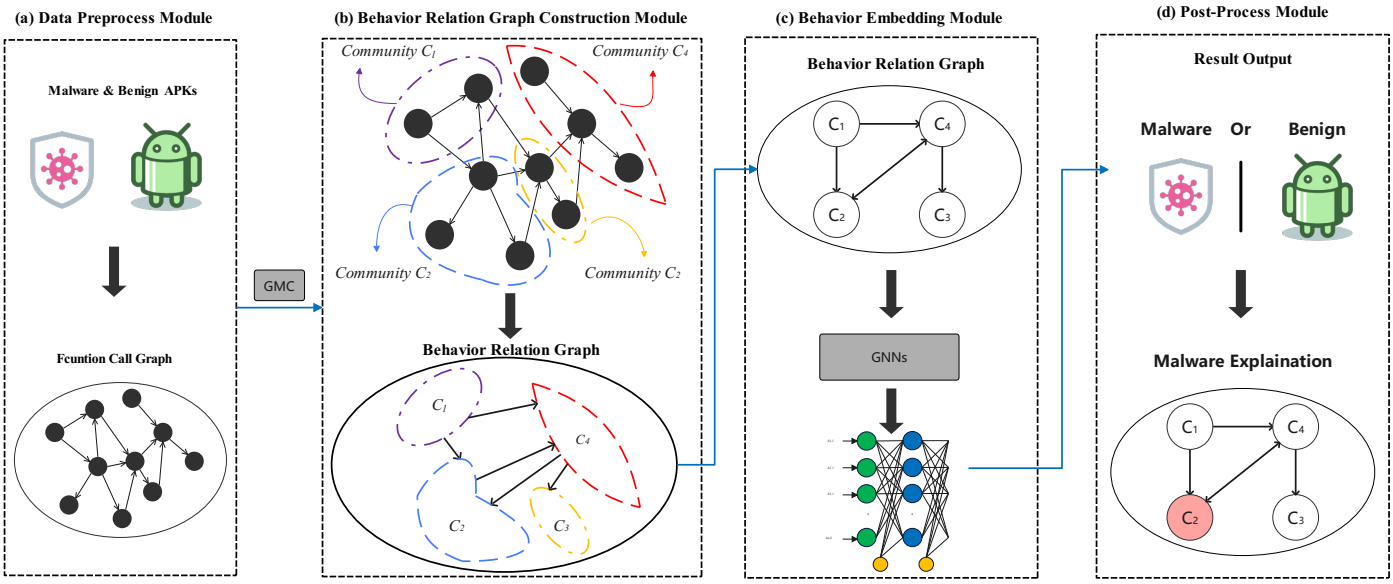
III. Methodology

Our overall method architecture is shown in Figure 1. The system architecture primarily includes the following key steps: 1) Data Preprocessing. 2) Community Graph Construction Module. 3) Behavior Embedding Module. 4) Post-Process Module.

A. Data Preprocess Module

The file preprocessing module performs file analysis and constructs the function call graph, transmitting the constructed

Fig. 1. The overview of GMCWare



function call graph to the next layer. First, using the Androguard library [29], APK files are statically analyzed to extract function call relationships. A directed graph from NetworkX [30] is utilized, where nodes represent functions and edges represent function call relationships, thereby constructing the Function Call Graph (FCG). Let **G = (V,E)**, where V represents functions and E represents function call relationships, as illustrated in part (a) of Figure 1.

B. Bahavior Relation Graph Construction Module

As shown in part (b) of Figure 1, The Community Function Call Graph Construction Module simplifies the upstream function call graph by processing all function nodes in the FCG, partitioning all functions into corresponding communities, and providing them to the downstream GNN Module.

Community Detection and Graph Simplification, The Greedy Modularity Optimization algorithm is applied to partition the function call graph into several communities $C_1, C_2, \ldots, C_k$, each representing a functional module or behavioral unit. The original nodes are mapped to community nodes, forming a simplified directed graph in which the nodes correspond to communities and the edges represent call relationships between communities. Community structure refers to the clustering phenomenon of nodes within the graph, where a subset of nodes is very tightly connected among themselves while having fewer connections with nodes in other parts. These tightly connected sets of nodes are referred to as communities. Modularity is a metric for measuring the quality of community partitioning in a graph, used to evaluate the superiority of a specific community partition relative to a random partition. A higher modularity value indicates a higher connection density within communities and fewer connections between communities, signifying a more reasonable partitioning. The definition of modularity is shown in Equation 1, where $A_{i,j}$ is the indicator function of whether there is an edge between node i and node j, k_i and k_j are the degrees of node i and node j respectively, m is the total number of edges in the graph, and $\delta(c_i, c_j)$ is the indicator function, which equals 1 if node i and node j belong to the same community, and 0 otherwise.

$$Q = \frac{1}{2m} \sum_{i,j} \left[A_{ij} - \frac{k_i k_j}{2m} \right] \delta(c_i, c_j) \tag{1}$$

We iteratively merge communities using the Greedy Modularity Optimization algorithm to progressively increase the modularity Q until no further improvement is possible. Specifically, the algorithm starts with each node forming its own community, calculates the incremental modularity change ΔQ for all possible pairs of communities if they were to be merged, selects the pair with the highest ΔQ to merge, and repeats this process until the modularity no longer increases. After completing community detection, the detected communities $C_1, C_2, \ldots, C_k$ are mapped to a simplified directed graph $G' = (V', E')$, where $V' = C_1, C_2, \ldots, C_k$, meaning that each community C_i corresponds to a supernode in the new graph G'. For each directed edge $(u, v) \in E$ in the original graph $G = (V, E)$, if node U belongs to community C_i and node and node v belongs to community C_j, and $i \neq j$, then a directed edge (C_i, C_j) is added to the simplified graph G'. Formally, the edge set E' is defined as shown in Equation2.

$$E' = \{ (C_i, C_j) \mid \exists\, u \in C_i, v \in C_j, (u, v) \in E \} \tag{2}$$

The entire process of the Greedy Modularity Optimization algorithm is shown in Algorithm 1. This process not only simplifies the graph structure and reduces complexity but also enhances the interpretability of the graph through a modular

view, making subsequent feature extraction and classification more efficient.

Algorithm 1: Greedy Modularity Communities Algorithm

Input : Unweighted graph $G = (V, E)$ where V is the set of nodes and E is the set of edges.
Output: Community structure of the graph G.

Initialize: Assign each node to its own community.
while *modularity can be increased* **do**
  for *each pair of communities* (C_i, C_j) **do**
    Compute the modularity gain ΔQ if C_i and C_j are merged.;
  end
  Identify the pair of communities (C_i^*, C_j^*) with the highest ΔQ.;
  if $\Delta Q > 0$ **then**
    Merge communities C_i^* and C_j^* into a new community C_{new}.;
  else
    Break the loop. No further modularity gain possible.;
  end
end
return *Final community structure.*

To transform the module call graph into feature vectors suitable for processing by graph neural network models, this paper employs a BERT [8]-based vectorization method. We extract features from the model nodes. If all data of each function within a community are used as the features of the module node, the graph structure data would become excessively bulky and computationally inefficient. Therefore, we extract the opcode and bytecode instructions for each function within each community. For external methods (External Method), we record the class name, method name, and description. For encoded methods (Encoded Method), we extract the bytecode instructions and their hexadecimal representations. To capture significant operations, the extracted instructions are categorized based on predefined classes, such as arithmetic operations, memory operations, control flow operations, and external calls, and the frequency of each type of instruction is counted. We use a pre-trained BERT [8] model to vectorize the descriptions of each community, generating 768-dimensional dense vectors. If a description is empty, a zero vector is used as padding. Finally, the aggregated clustering features of the instructions are converted into vectors and combined with the vectors generated by BERT [8] to form the final feature representation of the community nodes. This process is illustrated in Figure 2.

C. Behavior Embedding Module

The core idea of Graph Neural Networks (GNNs) is to transmit information through the structure of the graph, that

Fig. 2. community Node Embedding Generate

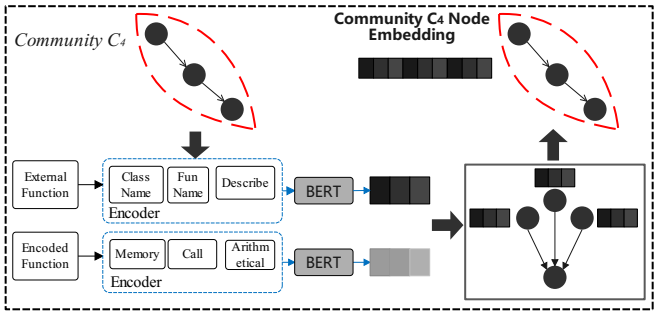
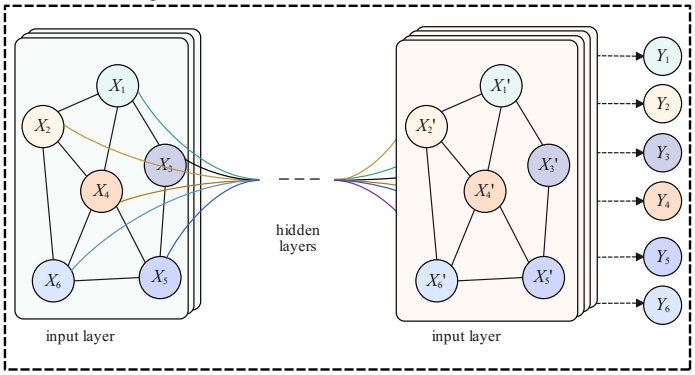


Fig. 3. The GNN module vectorizes the BRG



is, the connections between nodes. Graph Convolutional Networks (GCN) [31] are the simplest type of neural network models used for processing graph-structured data. Each node $v_i \in V$ in the graph has a feature vector $\mathbf{h}_i^{(0)}$, where $\mathbf{h}_i^{(0)} \in \mathbb{R}^d$ is the initial feature of node v_i, and d is the feature dimension. In GNNs, each node generates new information by aggregating the features of its neighboring nodes, combining the aggregated messages with the node's own features to create a new node representation. The update of the representation of node v_i can be expressed by Equation 4. Here: $\mathbf{h}_i^{(k)}$ is the representation of node v_i at layer k. $\mathcal{N}(i)$ denotes the set of neighboring nodes of node v_i. AGGREGATE is an aggregation operation, typically sum, mean, or max. $W^{(k)}$ is the weight matrix of layer k. $b^{(k)}$ is the bias term. σ is the activation function. In graph neural networks, the most commonly used aggregation methods include sum, mean, and max. In the final layer of a graph neural network, the output node representations can be used for different tasks. For the representation of the entire graph, it is usually necessary to aggregate the representations of all nodes (such as sum or mean) and then perform classification. This paper focuses on graph-level classification tasks, as shown in Equation 5.

Graph Neural Networks (GNNs) have many variants, each with different advantages in different tasks. GraphSAGE [32] introduced a mechanism for aggregating information through neighbor sampling to address computational issues on large-scale graphs. GAT [33] introduced the attention mechanism, using different weights in aggregating neighboring nodes. For each neighboring node $v_j \in \mathcal{N}(i)$, GAT computes its attention coefficient, as shown in Equation 3, which assigns different

weights to each neighboring node. TAGCN [34] introduced a topological attention mechanism to enhance the effectiveness of graph convolution by weighting and aggregating the topological structure of nodes.

$$\alpha_{ij} = \frac{\exp\left(\text{LeakyReLU}\left(\mathbf{a}^\top \left[W\mathbf{h}_i \,||\, W\mathbf{h}_j\right]\right)\right)}{\sum_{k\in\mathcal{N}(i)} \exp\left(\text{LeakyReLU}\left(\mathbf{a}^\top \left[W\mathbf{h}_i \,||\, W\mathbf{h}_k\right]\right)\right)} \tag{3}$$

$$\mathbf{h}_i^{(k+1)} = \sigma\left(W^{(k)} \cdot \text{AGGREGATE}\left(\{\mathbf{h}_j^{(k)} : j \in \mathcal{N}(i)\}\right) + b^{(k)}\right) \tag{4}$$

$$\hat{y} = \text{softmax}(W^{(L)} \cdot \text{AGGREGATE}(\{\mathbf{h}_i^{(L)} : i \in V\}) + b^{(L)}) \tag{5}$$

In this study, after constructing the BRG, we employed GNNs to represent the nodes and obtain feature vectors. This process is illustrated in Figure 3. GNNs learn the representations of nodes by aggregating information from their neighboring nodes.

D. Post-Process Module

GMCWare generates interpretative results for malware, primarily to facilitate reverse engineering. It elucidates the entire workflow of suspicious behaviors, providing interpretable outcomes for subsequent expert analysis. See the interpretability analysis for more results in the experimental sectionIV-D.

IV. RESULT AND DISCUSSION

A. Dataset And Metrics

The dataset used in this study consists of malicious executable files obtained from AndroZoo [35]. It includes 12,408 benign files (125 GB) and 7,591 malware families (80.558 GB). We disassembled the compiled executable files to obtain the corresponding Behavior Relation Graphs (BRGs).

To comprehensively evaluate our proposed method and measure the effectiveness of the GMC framework, we employ a series of widely used and representative performance metrics. These metrics include Accuracy, Precision, Recall, and F1 Score. The F1 Score is particularly useful in scenarios with class imbalance because it combines Precision and Recall, providing a more comprehensive assessment of the model's performance. The selection of these metrics aims to fully reflect the model's performance across different aspects, ensuring the comprehensiveness and reliability of the evaluation results. The specific definitions of these metrics and their application in this study are presented in Table I.

TABLE I
DESCRIPTIONS OF METRICS USED IN OUR EXPERIMENTS:

Merics	Abbr	Definition
True Positive	**TP**	The number of malware that is correctly classified as malware
True Negative	**TN**	The number of benign that is correctly classified as benign
False Positive	**FP**	The number of benign misclassified as malware
False Negative	**FN**	The number of malware misclassified as benign
True Positive Rate	**TPR**	$\text{TPR} = \frac{TP}{TP+FN}$
False Negative Rate	**FNR**	$\text{FNR} = \frac{FN}{TP+FN}$
True Negative Rate	**TNR**	$\text{TNR} = \frac{TN}{TN+FP}$
False Positive Rate	**FPR**	$\text{FPR} = \frac{FP}{TN+FP}$
Accuracy	**Acc**	$\text{Acc} = \frac{TP+TN}{TP+TN+FP+FN}$
Precision	**Pre**	$\text{Pre} = \frac{TP}{TP+FP}$
Recall	**Rec**	$\text{Rec} = \frac{TP}{TP+FN}$
F1-Socre	**F1**	$\text{F1} = \frac{2TP}{2TP+FP+FN}$

B. Simplification Effect

We use the GMC algorithm to trim the original function call graph (FCG), greatly simplifying its scale. The results, as shown in Table II, Table III, reveal that the size of the Function Call Graph (FCG) is much larger, with the number of nodes and edges significantly higher than that of the Behavior Relation Graph (BRG). For example, in benign samples, the average number of nodes in the Function Call Graph is 44,704, and the number of edges is 123,582, while the Behavior Relation Graph has an average of only 466 nodes and 1,091 edges. This difference indicates that the Function Call Graph is built from the perspective of program function calls, providing a finer granularity, hence a larger graph size. In contrast, the Behavior Relation Graph is more abstract, retaining only the relationships at the behavioral level, which significantly reduces the graph's size. As shown in Figure 4, we visualized the FCG and BRG before and after simplification for the APK file (with sha256 end is *****868). The left image shows the FCG, where the number of nodes and edges is large and complex (the edges overlap, appearing as gray areas in the image). The right image shows the simplified BRG, where the call structure is clear and the behavioral relationships are more apparent.

It is worth noting that we utilize BERT [8] to embed the vectorized features into the graph and store them. Nevertheless, the storage space required for the Function Call Graph remains considerably higher than that of the Behavior Relation Graph. For instance, in malware samples, the Function Call Graph occupies 203.3 GB, whereas the Behavior Relation Graph only occupies 4.28 GB. This demonstrates that the Behavior Relation Graph effectively reduces data size through simplification or aggregation, thereby significantly saving storage resources.

The Behavior Relation Graph is constructed from a higher-level abstraction of behavioral relationships, focusing on interactions between behaviors while ignoring specific call details. Therefore, it is more suitable for summarizing and analyzing behavioral patterns.

C. Discriminative power analysis

To better validate our proposed GMCWare, we selected several previously high-performing methods as baselines for our experiments. These include Drebin [5], NATICUSdroid [4], and the method proposed by Vinayaka K *et al.* [36].

NATICUSdroid [4], This method analyzes local permissions and custom permissions, selecting those that significantly con-

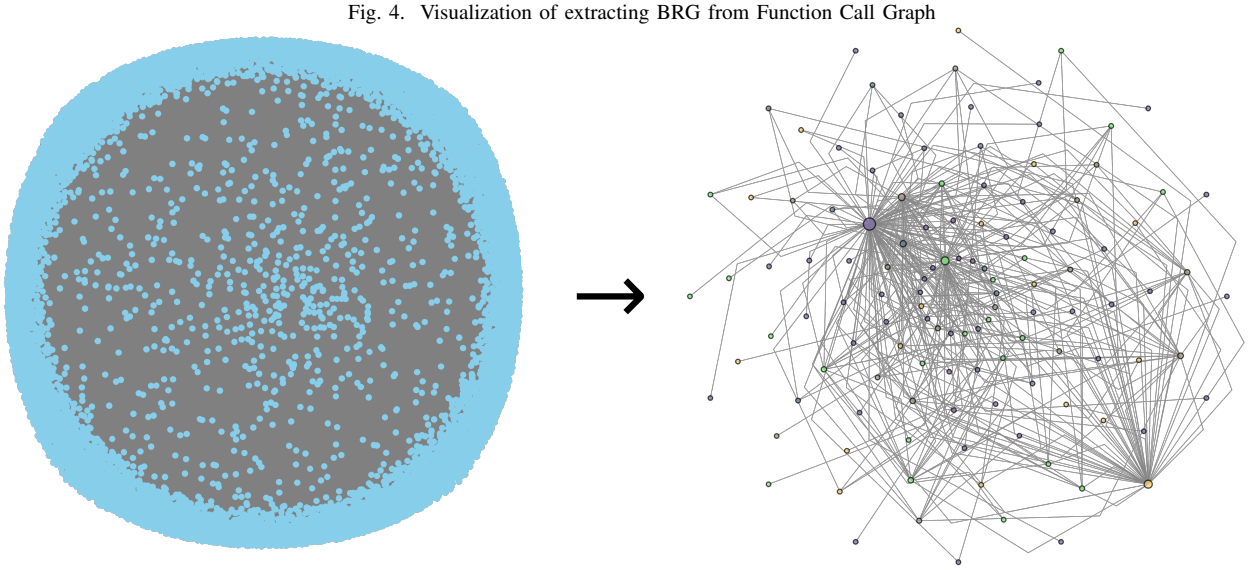
TABLE II
FUNCTION CALL GRAPH STATISTIC INFOMATION

Class	Apps	Function Call Graph						
		# Graph	Avg # Nodes	Avg # Edges	Median # Nodes	Median # Edges	Avg # Degree	Storage Space
Benign	12408	12162	44704	123582	39387	105245	3	533.6GB
Malware	7658	7591	24012	72157	16335	47545	3	203.3GB

TABLE III
BEHAVIOR RELATION GRAPH STATISTIC INFOMATION

Class	Apps	Behavior Relation Graph						
		# Graph	Avg # Nodes	Avg # Edges	Median # Nodes	Median # Edges	Avg # Degree	Storage Space
Benign	12408	12162	466	1091	418	1004	2	16.55GB
Malware	7658	7591	192	495	122	356	2	4.28GB

Fig. 4. Visualization of extracting BRG from Function Call Graph



tribute to detection. It creates a permission list by discarding irrelevant and non-essential permissions from the dataset. Based on this list, malware detection can be performed, thereby improving the malware detection rate.

DREBIN [5] is a lightweight Android malware detection method. It collects as many application features as possible, such as permissions and bytecode, embeds these features into a unified vector space, and finally classifies them using linear methods. DREBIN also offers a certain degree of interpretability.

Vinayaka K *et al.* [36] proposed a function-based Graph Convolutional Networks (GCNs) Android malware detection model utilizing the Function Call Graph (FCG). FCG captures the caller-callee relationships between internal methods of APKs in the form of a directed graph. Each node in the FCG is assigned a feature vector to represent its characteristics.

In our experiments, we compared the proposed model with traditional deep learning methods. The results show that the proposed model significantly outperforms these traditional methods in all metrics. First, in the NATICUSdroid [4] method, the traditional machine learning approaches such as SVM, Random Forest, and XGBoost achieve accuracies of 96.02%, 97.65%, and 95.95%, respectively. Although these methods perform well in terms of accuracy, they still exhibit certain false positive rates (FPR) and false negative rates (FNR). In contrast, the proposed model achieves 100% accuracy on this method, with both FPR and FNR equal to 0, completely eliminating false positives and false negatives, thus demonstrating its remarkable detection capability.

In the Drebin [5] method, the traditional SVM model attains an accuracy of 97.26%, with a precision of 97.13% and a recall of 97.54%. Its FPR is 1.90% and FNR is 2.46%. However, for the proposed model, whether using GAT or GCN, the accuracy reaches 99.10% and 100%, respectively, and both FPR and FNR are significantly reduced, further validating the superiority of our approach.

In the FCG [36] method, deep learning models such as GCN [31], GraphSAGE [32], and TAGCN [34] achieve accuracies of 74.56%, 74.62%, and 73.92%, respectively, while maintaining relatively high FPR and FNR, indicating their limited performance when handling complex graph structures. Under the same method, however, the proposed model (GCN [31], GAT [33], GraphSAGE [32], and TAGCN [34]) all reach 100% accuracy, with FPR and FNR both equaling 0, showcasing the

TABLE IV
PERFORMANCE COMPARISON OF THE PROPOSED MODELS WITH DEEP LEARNING MODELS (%)

Method		Accuracy	Precision	Recall	F1-Score	FPR	FNR
NATICUSdroid [4]	SVM	96.02	95.38	94.76	95.07	3.12	5.24
	Random Forest	97.65	94.45	95.18	94.81	3.80	4.82
	XGBoost	95.95	94.74	95.29	95.01	3.60	4.71
Drebin [5]	SVM	97.26	97.13	97.54	97.33	1.90	2.46
FCG [36]	GCN	74.56	84.80	46.40	59.98	5.80	53.60
	GraphSAGE	74.62	84.70	46.69	60.19	5.88	53.31
	TAGCN	73.92	84.72	45.50	59.20	5.83	55.50
Ours	GCN	**100.00**	**100.00**	**100.00**	**100.00**	**0.00**	**0.00**
	GAT	**99.10**	**99.90**	**99.62**	**99.76**	**0.00**	**0.38**
	GraphSAGE	**100.00**	**100.00**	**100.00**	**100.00**	**0.00**	**0.00**
	TAGCN	**100.00**	**100.00**	**100.00**	**100.00**	**0.00**	**0.00**

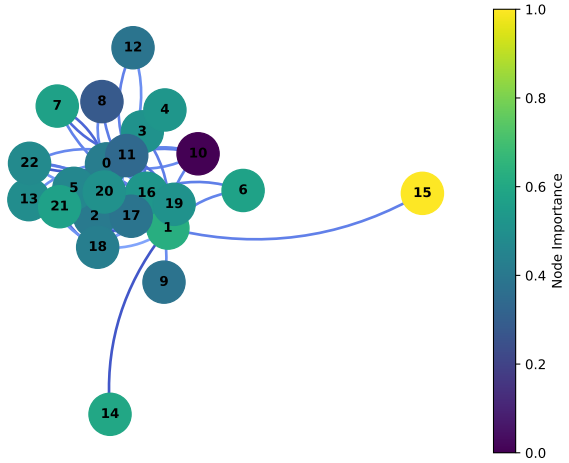


Fig. 5. Interpretability visualization of BRG

powerful capability of our approach in dealing with complex graph data.

In summary, the proposed model demonstrates outstanding performance across all methods, surpassing traditional methods in accuracy, precision, recall, and F1-score. Additionally, it achieves optimal control over both false positives and false negatives, highlighting its superiority in practical applications, particularly in fields such as malware detection.

D. Interpretability Via Subgraph Extraction

To verify the interpretability of our model, we employed the GNNExplainer [37] technique to explain the malware classification task based on GNN models. In the interpretability analysis, we further understood the model's decision-making process by calculating the importance of nodes. Node importance is obtained by multiplying the node features with the corresponding feature mask. In this step, we extracted the indices of the most important nodes and their corresponding feature values, calculating the feature importance for each node and identifying the most important node (i.e., the node with the highest feature importance value). By outputting the index, importance value, and relevant node features of this node, we help identify the nodes that have the greatest impact on the model's decision-making. To more intuitively demonstrate the model's decision-making process, we used graph visualization methods to visualize node importance and edge importance through color mapping, thereby helping us understand which nodes and edges have the greatest impact on the model's decision-making and assisting in malware decision-making, as shown in Figure 5. The color intensity of the nodes reflects their importance, and the color intensity of the edges reflects their importance. In this way, we can not only understand the most critical nodes but also intuitively see how the model makes decisions based on the information in the graph structure.

Experiments demonstrate that graph neural networks rely on key features of specific nodes when performing malware classification, and these features can be effectively interpreted and visualized, further enhancing the model's interpretability and transparency.

V. CONLUSION

In GMCWare, we introduced the concept of the Behavior Relation Graph (BRG), exploring its applicability in Android malware detection. BRG focuses on the relationships between behaviors, significantly reducing the size of the graph and the storage space required for graph data through algorithmic simplification, while retaining key behavioral features. This makes it suitable for Android malware detection tasks. Experimental results have demonstrated that this algorithm possesses high discriminative capability, achieving nearly 100% accuracy and almost 0% false positive and false negative rates on our dataset. Additionally, GMCWare provides interpretability.

In the future, we will focus on developing a graph classification method to distinguish between benign software and various malware families, which can help counteract obfuscation and other adversarial techniques. This will facilitate a deeper understanding of the origins of executable programs and their associations with different malware families.

VI. ACKNOWLEDGMENT

This research has been partially funded by the Institute of Information Engineering, Chinese Academy of Science,

Project E4V01511G3, and is supported by the University of Chinese Academy of Science.

REFERENCES

[1] A.-T. T. I. I.-S. Institute, "AV-ATLAS - Malware & PUA — portal.av-atlas.org," https://portal.av-atlas.org/malware/statistics, [Accessed 22-01-2025].

[2] L. Meijin, F. Zhiyang, W. Junfeng, C. Luyu, Z. Qi, Y. Tao, W. Yinwei, and G. Jiaxuan, "A systematic overview of android malware detection," *Applied Artificial Intelligence*, vol. 36, pp. 1–33, 12 2021.

[3] C. Li, Q. Lv, N. Li, Y. Wang, D. Sun, and Y. Qiao, "A novel deep framework for dynamic malware detection based on api sequence intrinsic features," *Computers & Security*, vol. 116, p. 102686, 2022.

[4] A. Mathur, L. M. Podila, K. Kulkarni, Q. Niyaz, and A. Y. Javaid, "Naticusdroid: A malware detection framework for android using native and custom permissions," *Journal of Information Security and Applications*, vol. 58, p. 102696, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2214212620308437

[5] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, and K. Rieck, "Drebin: Effective and explainable detection of android malware in your pocket," 02 2014.

[6] Y. He, Y. Liu, L. Wu, Z. Yang, K. Ren, and Z. Qin, "Msdroid: Identifying malicious snippets for android malware detection," *IEEE Transactions on Dependable and Secure Computing*, vol. 20, no. 3, pp. 2025–2039, 2023.

[7] M. Saqib, B. C. M. Fung, P. Charland, and A. Walenstein, "GAGE: Genetic algorithm-based graph explainer for malware analysis," in *Proc. of the 40th IEEE International Conference on Data Engineering (ICDE)*. Utrecht, Netherlands: IEEE Computer Society, May 2024, pp. 2258–2270.

[8] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[9] T. Sun, K. Allix, K. Kim, X. Zhou, D. Kim, D. Lo, T. F. Bissyandé, and J. Klein, "Dexbert: effective, task-agnostic and fine-grained representation learning of android bytecode," *IEEE Transactions on Software Engineering*, 2023.

[10] J. Huang, S. Han, W. You, W. Shi, B. Liang, J. Wu, and Y. Wu, "Hunting vulnerable smart contracts via graph embedding based bytecode matching," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 2144–2156, 2021.

[11] W. Wang, X. Wang, D. Feng, J. Liu, Z. Han, and X. Zhang, "Exploring permission-induced risk in android applications for malicious application detection," *IEEE Transactions on Information Forensics and Security*, vol. 9, no. 11, pp. 1869–1882, 2014.

[12] J. Tang, R. Li, Y. Jiang, X. Gu, and Y. Li, "Android malware obfuscation variants detection method based on multi-granularity opcode features," *Future Generation Computer Systems*, vol. 129, pp. 141–151, 2022.

[13] H. Cai, N. Meng, B. Ryder, and D. Yao, "Droidcat: Effective android malware detection and categorization via app-level profiling," *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 6, pp. 1455–1470, 2018.

[14] C. Rong, G. Gou, C. Hou, Z. Li, G. Xiong, and L. Guo, "Umvd-fsl: Unseen malware variants detection using few-shot learning," in *2021 international joint conference on neural networks (IJCNN)*. IEEE, 2021, pp. 1–8.

[15] P. Xu, C. Eckert, and A. Zarras, "hybrid-falcon: Hybrid pattern malware detection and categorization with network traffic and program code," *arXiv preprint arXiv:2112.10035*, 2021.

[16] F. Wei, S. Roy, X. Ou, and Robby, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps," *ACM Transactions on Privacy and Security (TOPS)*, vol. 21, no. 3, pp. 1–32, 2018.

[17] A. De Paola, S. Gaglio, G. L. Re, and M. Morana, "A hybrid system for malware detection on big data," in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, 2018, pp. 45–50.

[18] S. Gupta, P. Bansal, and S. Kumar, "Ulbp-rf: A hybrid approach for malware image classification," in *2018 Fifth International Conference on Parallel, Distributed and Grid Computing (PDGC)*. IEEE, 2018, pp. 115–119.

[19] I. Santos, J. Devesa, F. Brezo, J. Nieves, and P. G. Bringas, "Opem: A static-dynamic approach for machine-learning-based malware detection," in *International joint conference CISIS'12-ICEUTE´ 12-SOCO´ 12 special sessions*. Springer, 2013, pp. 271–280.

[20] S. Chen, M. Xue, Z. Tang, L. Xu, and H. Zhu, "Stormdroid: A streaminglized machine learning-based system for detecting android malware," in *Proceedings of the 11th ACM on Asia conference on computer and communications security*, 2016, pp. 377–388.

[21] M. H. Nguyen, D. Le Nguyen, X. M. Nguyen, and T. T. Quan, "Auto-detection of sophisticated malware using lazy-binding control flow graph and deep learning," *Computers & Security*, vol. 76, pp. 128–155, 2018.

[22] K. Tian, D. Yao, B. G. Ryder, G. Tan, and G. Peng, "Detection of repackaged android malware with code-heterogeneity features," *IEEE Transactions on Dependable and Secure Computing*, vol. 17, no. 1, pp. 64–77, 2017.

[23] J. D. Herath, P. P. Wakodikar, P. Yang, and G. Yan, "Cfgexplainer: Explaining graph neural network-based malware classification from control flow graphs," in *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2022, pp. 172–184.

[24] Y. Aafer, W. Du, and H. Yin, "Droidapiminer: Mining api-level features for robust malware detection in android," in *Security and Privacy in Communication Networks: 9th International ICST Conference, SecureComm 2013, Sydney, NSW, Australia, September 25-28, 2013, Revised Selected Papers 9*. Springer, 2013, pp. 86–103.

[25] H. Kim, J. Kim, Y. Kim, I. Kim, K. J. Kim, and H. Kim, "Improvement of malware detection and classification using api call sequence alignment and visualization," *Cluster Computing*, vol. 22, pp. 921–929, 2019.

[26] M. Cai, Y. Jiang, C. Gao, H. Li, and W. Yuan, "Learning features from enhanced function call graphs for android malware detection," *Neurocomputing*, vol. 423, pp. 301–307, 2021.

[27] A. Clauset, M. E. Newman, and C. Moore, "Finding community structure in very large networks," *Physical Review E—Statistical, Nonlinear, and Soft Matter Physics*, vol. 70, no. 6, p. 066111, 2004.

[28] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2008, p. P10008, 2008. [Online]. Available: https://api.semanticscholar.org/CorpusID:334423

[29] "GitHub - androguard/androguard: Reverse engineering and pentesting for Android applications — github.com," https://github.com/androguard/androguard, [Accessed 07-06-2024].

[30] "GitHub - networkx/networkx: Network Analysis in Python — github.com," https://github.com/networkx/networkx, [Accessed 03-01-2025].

[31] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," 2017. [Online]. Available: https://arxiv.org/abs/1609.02907

[32] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *NIPS*, 2017.

[33] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph Attention Networks," *International Conference on Learning Representations*, 2018, accepted as poster. [Online]. Available: https://openreview.net/forum?id=rJXMpikCZ

[34] J. Du, S. Zhang, G. Wu, J. M. F. Moura, and S. Kar, "Topology adaptive graph convolutional networks," 2018. [Online]. Available: https://arxiv.org/abs/1710.10370

[35] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting millions of android apps for the research community," in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16. New York, NY, USA: ACM, 2016, pp. 468–471. [Online]. Available: http://doi.acm.org/10.1145/2901739.2903508

[36] V. K. V and J. C. D, "Android malware detection using function call graph with graph convolutional networks," in *2021 2nd International Conference on Secure Cyber Computing and Communications (ICSCCC)*, 2021, pp. 279–287.

[37] Z. Ying, D. Bourgeois, J. You, M. Zitnik, and J. Leskovec, "Gnnexplainer: Generating explanations for graph neural networks," *Advances in neural information processing systems*, vol. 32, 2019.